



Informe Taller 2: Funciones de alto Orden

Jeidy Nicol Murillo Murillo - 235910
Verónica Lorena Mujica Gavidia - 2359406
Karol Tatiana Burbano Nasner - 2359305
Sebastian Castro Rengifo - 2359435

9 de noviembre de 2024

1. Informe de Procesos

En esta parte del informe se presentan los procesos generados por los programas recursivos, se generan algunos ejemplos para cada ejercicio, y muestre como se comporta el proceso generado por cada uno de los programas.

1.1. Funciones Básicas sobre Conjuntos Difusos

1.1.1. Función para conjunto difuso de números grandes

Esta función permite generar un conjunto difuso que evalúa el grado de pertenencia de un número n para representar qué tan grande es en función de los parámetros d (que controla la escala) y e (que ajusta la pendiente). La pertenencia se calcula como $(n/(n+d))^e$, donde valores más altos de n tendrán un mayor grado de pertenencia al conjunto "grande".

Ejemplo para `grande(10,3)`

```
1 test("Pertenencia de 5 en conjunto grande con d = 10 y e = 3") {  
2   val conjuntoGrande = objConjuntosDifusos.grande(10, 3)  
3   assert(conjuntoGrande(5) === Math.pow(5.0 / (5 + 10), 3))  
4 }
```

El test crea un conjunto difuso llamando a la función `grande` con los parámetros $d = 10$ y $e = 3$. Esto genera una instancia de `conjuntoGrande` que es una función que, para cualquier número n , calcula su pertenencia al conjunto difuso "grande" según la fórmula que definimos. En este punto, `grande` retorna la función `evaluarGrande`, que es una función que calcula el valor de pertenencia para un número dado n , en este caso $n = 5$.

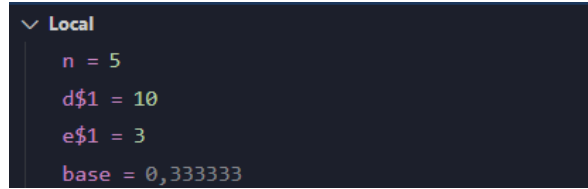
```
Local  
n = 5  
d$1 = 10  
e$1 = 3
```

Figura 1: Debug Grande

La primera verificación en `evaluarGrande` es si n es menor que 0. En este caso, $n = 5$, por lo que no se cumple $n < 0$, y continuamos con el cálculo.

La fórmula de pertenencia es:

$$\text{base} = \frac{n}{n + d} = \frac{5}{5 + 10} = \frac{5}{15} = 0,3333$$



```

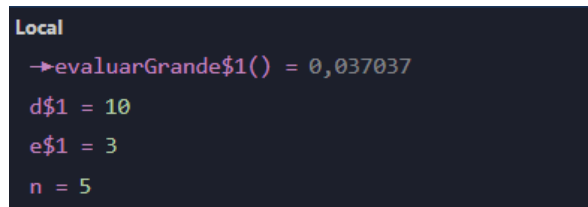
Local
  n = 5
  d$1 = 10
  e$1 = 3
  base = 0,333333

```

Figura 2: Debug Grande

Después, elevamos `base` a la potencia e (en este caso, $e = 3$):

$$\text{resultado} = \text{Math.pow}(\text{base}, e) = \text{Math.pow}(0,3333, 3) \approx 0,037$$



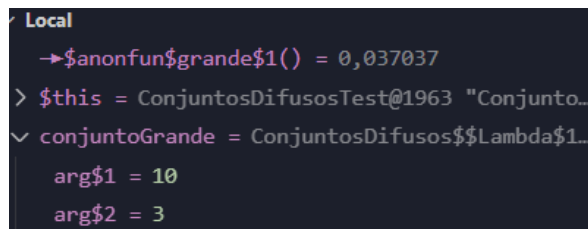
```

Local
  ->evaluarGrande$1() = 0,037037
  d$1 = 10
  e$1 = 3
  n = 5

```

Figura 3: Debug Grande

El test verifica que `conjuntoGrande(5)` (o `evaluarGrande(5)`) sea igual a `Math.pow(5.0 / (5 + 10), 3)`, que hemos calculado como aproximadamente 0,037. Como el cálculo cumple con esta igualdad, el `assert` pasa sin errores.

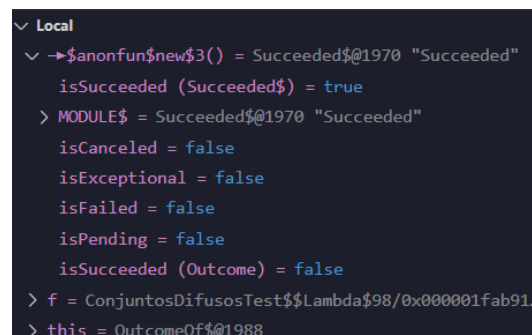


```

Local
  ->$anonfun$grande$1() = 0,037037
  > $this = ConjuntosDifusosTest@1963 "Conjunto..."
  > conjuntoGrande = ConjuntosDifusos$$Lambda$1...
  arg$1 = 10
  arg$2 = 3

```

Figura 4: Debug Grande



```

Local
  > ->$anonfun$new$3() = Succeeded$@1970 "Succeeded"
  isSucceeded (Succeeded$) = true
  > MODULE$ = Succeeded$@1970 "Succeeded"
  isCanceled = false
  isExceptional = false
  isFailed = false
  isPending = false
  isSucceeded (Outcome) = false
  > f = ConjuntosDifusosTest$$Lambda$98/0x000001fab91...
  > this = OutcomeOf$@1988

```

Figura 5: Debug Grande

1.1.2. Función Complemento

La función complemento toma un conjunto difuso c y devuelve un nuevo conjunto difuso que representa el complemento de c , calculando para cada valor x el grado de pertenencia como $1 - c(x)$. Esto permite crear conjuntos difusos complementarios, invirtiendo los valores de pertenencia del conjunto original.

Ejemplo para `complemento(7,3)`

```
1 test("Complemento de un conjunto con valores negativos debe ser 1") {
2   val conjunto = objConjuntosDifusos.grande(7, 3)
3   val complementoConj = objConjuntosDifusos.complemento(conjunto)
4
5   val valorComplemento = objConjuntosDifusos.pertenece(-1, complementoConj)
6   assert(Math.abs(valorComplemento - 1.0) < 0.0001)
7 }
```

El test crea un conjunto difuso llamando a la función `grande` con los parámetros $d = 7$ y $e = 3$, generando una instancia de `conjuntoGrande` que calcula la pertenencia al conjunto "grande" para cualquier número n según una fórmula definida. Luego, calcula el complemento del conjunto y evalúa la pertenencia de -1 en este complemento, esperando un valor cercano a 1. Si la diferencia es menor a 0,0001, el test es exitoso, confirmando el correcto funcionamiento de la función `complemento` con valores negativos.

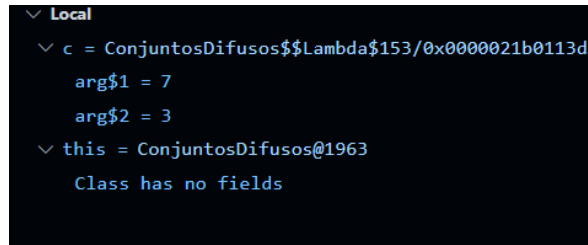


Figura 6: Debug Complemento

El debug inicia con la creación de la instancia de `ConjuntosDifusos` con los argumentos `arg$1 = 7` y `arg$2 = 3`, estableciendo el punto de partida para el cálculo del complemento.

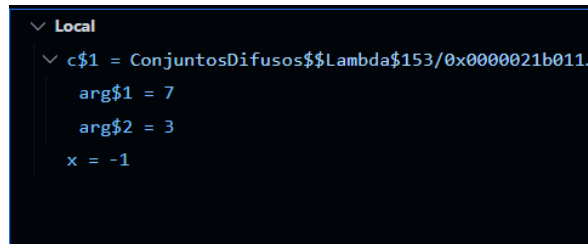


Figura 7: Debug Complemento

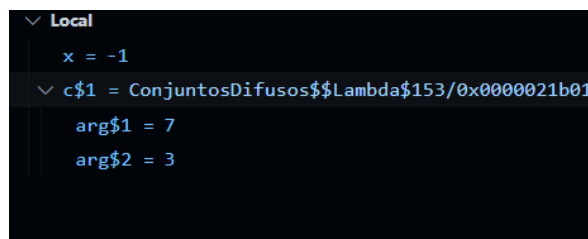


Figura 8: Debug Complemento

Un momento crucial es cuando `calcularComplemento$1()` comienza su ejecución, mostrando el valor 1.000000 que representa el complemento calculado exitosamente.

```

Local
└─> calcularComplemento() = ConjuntosDifusos$$Lambda$157/...
  └─arg$1 = ConjuntosDifusos$$Lambda$153/0x00000190...
    └─arg$1 = 7
    └─arg$2 = 3
  > $this = ComplementoTest@1972 "ComplementoTest"
  └─conjunto = ConjuntosDifusos$$Lambda$153/0x00000...
    └─arg$1 = 7
    └─arg$2 = 3

```

Figura 9: Debug Complemento

La función anónima `$anonfun$complemento$1()` procesa el cálculo manteniendo las referencias necesarias al conjunto original.

```

Local
└─> $this = ComplementoTest@1972 "ComplementoTest"
  └─conjunto = ConjuntosDifusos$$Lambda$153/0x000001...
    └─arg$1 = 7
    └─arg$2 = 3
  └─complementoConj = ConjuntosDifusos$$Lambda$157/0...
    └─arg$1 = ConjuntosDifusos$$Lambda$153/0x00000190...
      └─arg$1 = 7
      └─arg$2 = 3

```

Figura 10: Debug Complemento

```

Local
└─c$1 = ConjuntosDifusos$$Lambda$153/0x00000190d41...
  └─arg$1 = 7
  └─arg$2 = 3
  └─x = -1

```

Figura 11: Debug Complemento

```

Local
└─x = -1
  └─c$1 = ConjuntosDifusos$$Lambda$153/0x00000190d4...
    └─arg$1 = 7
    └─arg$2 = 3

```

Figura 12: Debug Complemento

```

▼ Local
  →calcularComplemento$1() = 1,000000
  ▼ c$1 = ConjuntosDifusos$$Lambda$153/0x00000190d41...
    arg$1 = 7
    arg$2 = 3
    x = -1

```

Figura 13: Debug Complemento

```

▼ Local
  →$anonfun$complemento$1() = 1,000000
  elem = -1
  ▼ s = ConjuntosDifusos$$Lambda$157/0x00000190d413f...
    ▼ arg$1 = ConjuntosDifusos$$Lambda$153/0x00000190...
      arg$1 = 7
      arg$2 = 3
  ▼ this = ConjuntosDifusos@1963
    Class has no fields

```

Figura 14: Debug Complemento

```

▼ Local
  →pertenece() = 1,000000
  > $this = ComplementoTest@1972 "ComplementoTest"
  ▼ conjunto = ConjuntosDifusos$$Lambda$153/0x000001...
    arg$1 = 7
    arg$2 = 3
  ▼ complementoConj = ConjuntosDifusos$$Lambda$157/0...
    ▼ arg$1 = ConjuntosDifusos$$Lambda$153/0x00000190...
      arg$1 = 7
      arg$2 = 3

```

Figura 15: Debug Complemento

```

▼ Local
  > $this = ComplementoTest@1972 "ComplementoTest"
  ▼ conjunto = ConjuntosDifusos$$Lambda$153/0x000001...
    arg$1 = 7
    arg$2 = 3
  ▼ complementoConj = ConjuntosDifusos$$Lambda$157/0...
    ▼ arg$1 = ConjuntosDifusos$$Lambda$153/0x00000190...
      arg$1 = 7
      arg$2 = 3
  valorComplemento = 1,000000

```

Figura 16: Debug Complemento

1.1.3. Función Unión

Esta función permite la unión de dos conjuntos difusos se define como el máximo grado de pertenencia entre ambos conjuntos.

Ejemplo de Unión de dos conjuntos distintos, grado máximo”

```
1 test("Unión de dos conjuntos distintos, grado máximo en ambos") {
2     val conjunto1 = objConjuntosDifusos.grande(3, 2)
3     val conjunto2 = objConjuntosDifusos.grande(5, 2)
4     val union = objConjuntosDifusos.union(conjunto1, conjunto2)
5     assert((0 to 1000).forall(n => union(n) == math.max(conjunto1(n), conjunto2(n)
6     )))
7 }
```

Paso 1: Preparación de los conjuntos difusos

val conjunto1 = objConjuntosDifusos.grande(3, 2) Aquí, se está llamando a un método grande del objeto objConjuntosDifusos para generar el primer conjunto difuso (conjunto1) con los parámetros:

- 3 como valor central (pico).
- 2 como grado de difusión máximo.

Esto define un conjunto difuso con una función de pertenencia que decrece en ambos lados del punto 3 y alcanza un grado máximo de 2.

- val conjunto2 = objConjuntosDifusos.grande(5, 2)

De manera similar, se crea el segundo conjunto difuso (conjunto2) con un valor central en 5 y el mismo grado de difusión máximo (2). Esto crea un conjunto difuso con un pico en 5 y el mismo comportamiento difuso que conjunto1.

Paso 2: Realización de la operación de unión

- val union = objConjuntosDifusos.union(conjunto1, conjunto2)

Se aplica la operación de unión a conjunto1 y conjunto2 mediante el método unión en objConjuntosDifusos. En teoría, la operación de unión entre conjuntos difusos se define como el máximo de los grados de pertenencia en cada punto. Por lo tanto, union debería devolver una nueva función de pertenencia donde, para cada valor de entrada n, la pertenencia será el valor máximo entre conjunto1(n) y conjunto2(n).

Paso 3: Verificación del resultado

- assert((0 to 1000).forall(n => union(n) == math.max(conjunto1(n), conjunto2(n))))

Esta línea asegura que para cada n en el rango de 0 a 1000, la pertenencia de union(n) sea el máximo entre los grados de pertenencia de conjunto1(n) y conjunto2(n).

Usa forall para verificar que esta condición se cumple en cada punto del intervalo. Si la condición es verdadera para todos los n, el test pasará; de lo contrario, fallará, indicando que la operación de unión no se implementó correctamente.

- En esta primera parte nos muestra la inicialización de conjunto1 y conjunto2. Aquí se verifica que cada conjunto difuso ha sido creado correctamente, asignando sus parámetros específicos de centro y difusión (por ejemplo, conjunto1 centrado en 3 con difusión 2 y conjunto2 en 5 con la misma difusión). Esta figura confirma que la estructura inicial de ambos conjuntos está lista para la prueba de unión.

```
▼ Local
  ▼ cd1 = ConjuntosDifusos$$Lambda$153/0x0000016ac613...
    arg$1 = 3
    arg$2 = 2
  ▼ cd2 = ConjuntosDifusos$$Lambda$153/0x0000016ac613...
    arg$1 = 5
    arg$2 = 2
  > this = ConjuntosDifusos@1961
```

Figura 17: Debug Unión

- La siguiente parte del test muestra el momento justo después de que se llama a la función `union`, la cual tomará `conjunto1` y `conjunto2` como parámetros para realizar la operación. Esta figura verifica que la función ha iniciado y ha recibido correctamente ambos conjuntos.

```
▼ Local
  ▼ →union() = ConjuntosDifusos$$Lambda$157/0x000001...
  ▼ arg$1 = ConjuntosDifusos$$Lambda$153/0x0000016a...
    arg$1 = 3
    arg$2 = 2
  ▼ arg$2 = ConjuntosDifusos$$Lambda$153/0x0000016a...
    arg$1 = 5
    arg$2 = 2
  > $this = UnionTest@1971 "UnionTest"
  ▼ conjunto1 = ConjuntosDifusos$$Lambda$153/0x000001...
    arg$1 = 3
    arg$2 = 2
  ▼ conjunto2 = ConjuntosDifusos$$Lambda$153/0x000001...
    arg$1 = 5
    arg$2 = 2
```

Figura 18: Debug Unión

- En esta parte del código se comienza a calcular la unión para el primer valor de `n`, verificando que `union(n)` devuelve el máximo valor de pertenencia entre `conjunto1(n)` y `conjunto2(n)`. Este paso establece que la lógica de `math.max` funciona correctamente al inicio del rango de prueba.

```
▼ Local
  ▼ cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  ▼ cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 0
```

Figura 19: Debug Unión

- Continúa el proceso de cálculo de la unión en los primeros valores de `n`. La función aplica `math.max` y confirma que el valor de pertenencia de `union(n)` coincide con el valor esperado, lo que demuestra que la función de unión devuelve el valor correcto en cada punto al comienzo de la iteración.

```

v Local
  n = 0
  v cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  v cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2

```

Figura 20: Debug Unión

- Continúa mostrando un paso intermedio en el que el valor de n incrementa y la función sigue calculando $\text{union}(n)$ como el máximo entre los valores de pertenencia de conjunto1 y conjunto2 . Esto confirma que el proceso sigue funcionando como se espera a medida que el valor de n avanza.

```

v Local
  -> $anonfun$union$1() = 0,000000
  v union$2 = ConjuntosDifusos$$Lambda$157/0x0000016a...
  v arg$1 = ConjuntosDifusos$$Lambda$153/0x0000016a...
    arg$1 = 3
    arg$2 = 2
  v arg$2 = ConjuntosDifusos$$Lambda$153/0x0000016a...
    arg$1 = 5
    arg$2 = 2
  > conjunto1$1 = ConjuntosDifusos$$Lambda$153/0x0000...
  > conjunto2$1 = ConjuntosDifusos$$Lambda$153/0x0000...
  n = 0

```

Figura 21: Enter Caption

- Ilustra el cálculo de unión para valores de n mayores. La función sigue aplicando math.max en cada punto y confirma que $\text{union}(n)$ devuelve el valor máximo en cada caso. Esta figura verifica que el proceso se mantiene consistente en puntos más avanzados.

```

v Local
  v cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  v cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 1

```

Figura 22: Debug Unión

- A medida que la prueba continúa, esta figura demuestra que la función union sigue devolviendo el máximo valor entre $\text{conjunto1}(n)$ y $\text{conjunto2}(n)$ en cada nuevo valor de n . Esta figura da continuidad al proceso de verificación para valores medianos de n .


```
Local
  →calcularUnion$1() = 0,062500
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 1
```

Figura 23: Debug Unión

- En esta imagen se muestra el cálculo de unión en una etapa más avanzada del rango. La función `union` sigue funcionando correctamente, calculando el valor de pertenencia más alto entre los dos conjuntos. Esta figura refuerza que `union` cumple su función teórica en cada iteración.

```
Local
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 2
```

Figura 24: Debug Unión

- En este punto, el valor de `n` sigue incrementando, y la función continúa verificando que `union(n)` es igual a `math.max(conjunto1(n), conjunto2(n))`. Esta figura confirma que no hay errores en el cálculo para valores de `n` más altos en el rango.

```
Local
  →calcularUnion$1() = 0,160000
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 2
```

Figura 25: Debug Unión

- Esta figura muestra una etapa intermedia donde el cálculo de unión sigue correcto para cada valor de `n`, reafirmando que el valor de pertenencia en `union` es el máximo entre ambos conjuntos. Es otra validación de consistencia en el proceso.

```
Local
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 3
```

Figura 26: Debug Unión

- En esta parte se verifica nuevamente el cálculo de unión en el siguiente conjunto de valores de n , demostrando que `union(n)` sigue devolviendo el valor correcto entre `conjunto1` y `conjunto2`. Esto muestra que la función mantiene su precisión a lo largo del dominio.

```
Local
  →calcularUnion$1() = 0,250000
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 3
```

Figura 27: Debug Unión

- Muestra un paso más en la secuencia, donde el cálculo de `union` se realiza de forma correcta en valores altos de n . La figura indica que el comportamiento de la función es estable incluso en esta etapa avanzada.

```
Local
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 4
```

Figura 28: Debug Unión

- En la siguiente parte del test presenta el cálculo de unión hacia el final del rango. La función `union(n)` sigue devolviendo `math.max(conjunto1(n), conjunto2(n))` como se espera, lo cual es una verificación importante antes de concluir la prueba.

```
Local
  ->calcularUnion$1() = 0,326531
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 4
```

Figura 29: Debug Unión

- En esta parte se muestra la consistencia del cálculo de **union** en los valores finales de **n**, comprobando que la función sigue ejecutando la operación correctamente en los límites superiores del rango de prueba.

```
Local
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 50
```

Figura 30: Debug Unión

- En esta parte se acerca a los últimos valores en el rango de prueba, donde **union(n)** continúa devolviendo el máximo valor de pertenencia en cada punto, reafirmando el correcto funcionamiento de la función en esta etapa avanzada.

```
Local
  ->calcularUnion$1() = 0,889996
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 3
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000016ac6...
    arg$1 = 5
    arg$2 = 2
  n = 50
```

Figura 31: Debug Unión

- La función se aproxima al final del rango y sigue calculando **union(n)** como el máximo entre **conjunto1(n)** y **conjunto2(n)**. Esta figura confirma la estabilidad de la operación de unión.

```
▼ Local
  ▼ cd1$1 = ConjuntosDifusos$$Lambda$153/0x000001c401...
    arg$1 = 3
    arg$2 = 2
  ▼ cd2$1 = ConjuntosDifusos$$Lambda$153/0x000001c401...
    arg$1 = 5
    arg$2 = 2
  n = 200
```

Figura 32: Debug Unión

- En los últimos valores de prueba, la figura muestra que la función `union` sigue devolviendo valores de pertenencia esperados, confirmando una vez más que `math.max` se aplica sin errores en esta última fase.

```
▼ Local
  →calcularUnion$1() = 0,970662
  ▼ cd1$1 = ConjuntosDifusos$$Lambda$153/0x000001c401...
    arg$1 = 3
    arg$2 = 2
  ▼ cd2$1 = ConjuntosDifusos$$Lambda$153/0x000001c401...
    arg$1 = 5
    arg$2 = 2
  n = 200
```

Figura 33: Debug Unión

- Estas figuras finales representan los últimos pasos del cálculo en el rango completo de prueba (hasta $n=1000$). La función `union` termina de evaluar los valores en el dominio y confirma que `union(n)` es el valor máximo entre `conjunto1(n)` y `conjunto2(n)`. Las siguientes imágenes muestran y aseguran que la implementación es robusta en todo el dominio, y no presenta errores.

```
▼ Local
  ▼ cd1$1 = ConjuntosDifusos$$Lambda$153/0x000001fe81...
    arg$1 = 3
    arg$2 = 2
  ▼ cd2$1 = ConjuntosDifusos$$Lambda$153/0x000001fe81...
    arg$1 = 5
    arg$2 = 2
  n = 1000
```

Figura 34: Debug Unión

```

v Local
  -> calcularUnion$1() = 0,994027
  v cd1$1 = ConjuntosDifusos$$Lambda$153/0x000001fe81...
    arg$1 = 3
    arg$2 = 2
  v cd2$1 = ConjuntosDifusos$$Lambda$153/0x000001fe81...
    arg$1 = 5
    arg$2 = 2
  n = 1000

```

Figura 35: Debug Unión

```

v Local
  v -> apply() = Boolean@2073 "true"
  > FALSE = Boolean@2080 "false"
    serialVersionUID = -3665804199014368530
  > TRUE = Boolean@2073 "true"
  > TYPE = Class (boolean)@2081 "boolean"
    value = true
  v p = UnionTest$$Lambda$158/0x000001fe8113ea78@2054...
    > arg$1 = ConjuntosDifusos$$Lambda$157/0x000001fe...
    > arg$2 = ConjuntosDifusos$$Lambda$153/0x000001fe...
    > arg$3 = ConjuntosDifusos$$Lambda$153/0x000001fe...
    res = true
  > it = RangeIterator@2074 "<iterator>"
  > this = Range$Inclusive@2075 "Range 0 to 1000"

```

Figura 36: Debug Unión

```

v Local
  -> forall() = true
  > $this = UnionTest@2094 "UnionTest"
  v conjunto1 = ConjuntosDifusos$$Lambda$153/0x000001...
    arg$1 = 3
    arg$2 = 2
  v conjunto2 = ConjuntosDifusos$$Lambda$153/0x000001...
    arg$1 = 5
    arg$2 = 2
  > union = ConjuntosDifusos$$Lambda$157/0x000001fe81...

```

Figura 37: Debug Unión

```

v Local
  v -> runTest() = SucceededStatus$@2291
  > MODULE$ = SucceededStatus$@2291
  > $this = UnionTest@2094 "UnionTest"
  > testName = "Unión de dos conjuntos distintos, grado máximo en ambos"
  > args = Args@2118 "Args(org.scalatest.WrapperCatchReporter@71f0b56c,or..."

```

Figura 38: Debug Unión

Cada figura presentado anteriormente contribuye a verificar que la función `union` cumple con la operación teórica de tomar el valor máximo entre dos conjuntos difusos en cada punto del dominio. Las figuras 17 hasta la figura 38 en conjunto validan que, desde el inicio hasta el final del rango, `union` se comporta de acuerdo con lo esperado en el cálculo de pertenencia máxima entre `conjunto1` y `conjunto2`.

1.1.4. Función Intersección

Esta función permite la intersección de dos conjuntos difusos se define como el mínimo grado de pertenencia entre ambos conjuntos. La función recibe dos conjuntos difusos (`cd1` y `cd2`) y genera una función `calcularInterseccion` que calcula el mínimo de los grados de pertenencia para un elemento en ambos conjuntos. Al retornar `calcularInterseccion`, estamos devolviendo una función que, cuando se llama con un índice `n`, aplica `math.min` para obtener el grado de pertenencia en la intersección.

Ejemplo de Intersección de dos conjuntos distintos, grado mínimo en ambos

```
1 test("Intersección de dos conjuntos distintos, grado mínimo en ambos") {
2     val conjunto1 = objConjuntosDifusos.grande(3, 2)
3     val conjunto2 = objConjuntosDifusos.grande(5, 2)
4     val interseccion = objConjuntosDifusos.interseccion(conjunto1, conjunto2)
5     assert((0 to 1000).forall(n => interseccion(n) == math.min(conjunto1(n),
6         conjunto2(n))))
7 }
```

En este test dado:

- `conjunto1` es un conjunto difuso generado con `grande(3, 2)`.
- `conjunto2` es otro conjunto difuso generado con `grande(5, 2)`.
- La intersección se valida verificando que, para cada elemento `n` de 0 a 1000, el valor en la intersección sea `math.min(conjunto1(n), conjunto2(n))`.

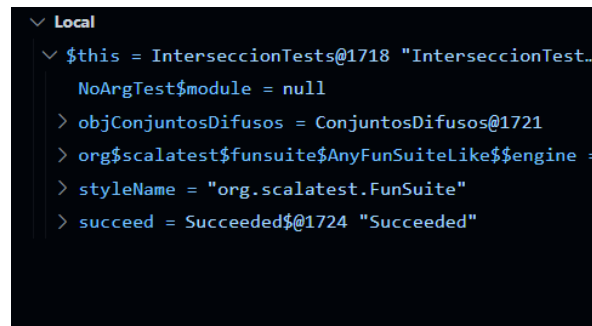


Figura 39: Debug Intersección

En esta primera imagen se puede observar la inicialización de los conjuntos difusos con sus parámetros:

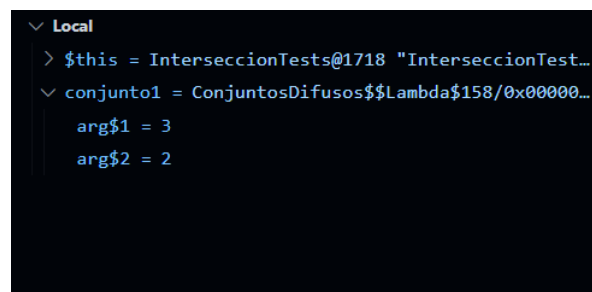


Figura 40: Debug Intersección

```

Local
> $this = InterseccionTests@1718 "InterseccionTest..."
conjunto1 = ConjuntosDifusos$$Lambda$158/0x000000...
    arg$1 = 3
    arg$2 = 2
conjunto2 = ConjuntosDifusos$$Lambda$158/0x000000...
    arg$1 = 5
    arg$2 = 2

```

Figura 41: Debug Intersección

```

Local
> $this = InterseccionTests@1718 "InterseccionTest..."
conjunto1 = ConjuntosDifusos$$Lambda$158/0x000000...
    arg$1 = 3
    arg$2 = 2
conjunto2 = ConjuntosDifusos$$Lambda$158/0x000000...
    arg$1 = 5
    arg$2 = 2
interseccion = ConjuntosDifusos$$Lambda$159/0x00...
    arg$1 = ConjuntosDifusos$$Lambda$158/0x000001c1...
        arg$1 = 3
        arg$2 = 2
    arg$2 = ConjuntosDifusos$$Lambda$158/0x000001c1...
        arg$1 = 5
        arg$2 = 2

```

Figura 42: Debug Intersección

Esta imagen muestra el estado inicial de la prueba donde:

- $n = 0$: Punto de inicio de la iteración
- $cd1\$2$ y $cd2\$2$ mantienen los parámetros originales

```

Local
interseccion$2 = ConjuntosDifusos$$Lambda$159/0x...
    arg$1 = ConjuntosDifusos$$Lambda$158/0x000001c1...
        arg$1 = 3
        arg$2 = 2
    arg$2 = ConjuntosDifusos$$Lambda$158/0x000001c1...
        arg$1 = 5
        arg$2 = 2
conjunto1$1 = ConjuntosDifusos$$Lambda$158/0x000...
    arg$1 = 3
    arg$2 = 2
conjunto2$1 = ConjuntosDifusos$$Lambda$158/0x000...
    arg$1 = 5
    arg$2 = 2
n = 0

```

Figura 43: Debug Intersección

```
Local
n = 0
cd1$2 = ConjuntosDifusos$$Lambda$158/0x000001c10...
  arg$1 = 3
  arg$2 = 2
cd2$2 = ConjuntosDifusos$$Lambda$158/0x000001c10...
  arg$1 = 5
  arg$2 = 2
```

Figura 44: Debug Intersección

Durante el proceso de iteración, se observa cómo la variable n va incrementando mientras se mantienen constantes los parámetros de los conjuntos difusos, permitiendo evaluar la intersección en diferentes puntos del dominio.

```
Local
interseccion$2 = ConjuntosDifusos$$Lambda$159/0x...
  arg$1 = ConjuntosDifusos$$Lambda$158/0x000001c1...
    arg$1 = 3
    arg$2 = 2
  arg$2 = ConjuntosDifusos$$Lambda$158/0x000001c1...
    arg$1 = 5
    arg$2 = 2
conjunto1$1 = ConjuntosDifusos$$Lambda$158/0x000...
  arg$1 = 3
  arg$2 = 2
conjunto2$1 = ConjuntosDifusos$$Lambda$158/0x000...
  arg$1 = 5
  arg$2 = 2
n = 1
```

Figura 45: Debug Intersección

```
Local
n = 1
cd1$2 = ConjuntosDifusos$$Lambda$158/0x000001c10...
  arg$1 = 3
  arg$2 = 2
cd2$2 = ConjuntosDifusos$$Lambda$158/0x000001c10...
  arg$1 = 5
  arg$2 = 2
```

Figura 46: Debug Intersección


```

Local
  interseccion$2 = ConjuntosDifusos$$Lambda$159/0x...
  arg$1 = ConjuntosDifusos$$Lambda$158/0x000001c1...
    arg$1 = 3
    arg$2 = 2
  arg$2 = ConjuntosDifusos$$Lambda$158/0x000001c1...
    arg$1 = 5
    arg$2 = 2
  conjunto1$1 = ConjuntosDifusos$$Lambda$158/0x000...
    arg$1 = 3
    arg$2 = 2
  conjunto2$1 = ConjuntosDifusos$$Lambda$158/0x000...
    arg$1 = 5
    arg$2 = 2
  n = 2

```

Figura 47: Debug Intersección

```

Local
  interseccion$2 = ConjuntosDifusos$$Lambda$159/0x...
  arg$1 = ConjuntosDifusos$$Lambda$158/0x000001d7...
    arg$1 = 3
    arg$2 = 2
  arg$2 = ConjuntosDifusos$$Lambda$158/0x000001d7...
    arg$1 = 5
    arg$2 = 2
  conjunto1$1 = ConjuntosDifusos$$Lambda$158/0x000...
    arg$1 = 3
    arg$2 = 2
  conjunto2$1 = ConjuntosDifusos$$Lambda$158/0x000...
    arg$1 = 5
    arg$2 = 2
  n = 1000

```

Figura 48: Debug Intersección

```

Local
  n = 1000
  cd1$2 = ConjuntosDifusos$$Lambda$158/0x000001d7a...
    arg$1 = 3
    arg$2 = 2
  cd2$2 = ConjuntosDifusos$$Lambda$158/0x000001d7a...
    arg$1 = 5
    arg$2 = 2

```

Figura 49: Debug Intersección

En el estado final ($n = 1000$) se ha completado la verificación de la intersección en todo el rango especificado. Los resultados confirman que:

- Los parámetros de los conjuntos se mantuvieron constantes durante toda la ejecución

- La función de intersección evaluó correctamente el mínimo entre ambos conjuntos
- La implementación fue robusta a lo largo de todo el dominio de prueba

Las demás imágenes de depuración muestran estados intermedios del proceso de iteración que siguen el mismo patrón de comportamiento, validando la correcta implementación de la función de intersección de conjuntos difusos.

1.1.5. Función Inclusión

La función inclusion evalúa si todos los elementos del conjunto difuso cd1 están incluidos en cd2 mediante un recorrido de cada elemento hasta el índice 1000. Esto se realiza de manera recursiva a través de la función loop, que es una función recursiva de cola (tail recursion). La anotación @tailrec optimiza la recursión para que no se acumulen llamadas en la pila, logrando que solo se mantenga una llamada activa a la vez.

```
1 test("Conjunto difuso incluido en si mismo") {
2     val conjunto = objConjuntosDifusos.grande(4, 2)
3     assert(objConjuntosDifusos.inclusion(conjunto, conjunto))
4 }
```

Si, para algún valor s , la membresía de $cd1(s)$ es mayor que la de $cd2(s)$, la función devuelve false, indicando que $cd1$ no está incluido en $cd2$. Si se recorre todo el rango sin encontrar un valor que viole la inclusión, la función retorna true.

Ejemplo para `inclusion(4, 2)`

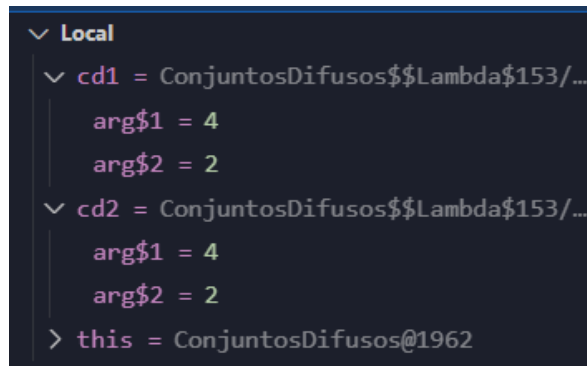


Figura 50: Debug inclusión

Primera llamada a loop ($i = 0$). La recursión de cola loop se inicia con el índice $i = 0$. Condición $if(i > 1000) : i$ es igual a 0, así que esta condición es falsa, y la función continúa.

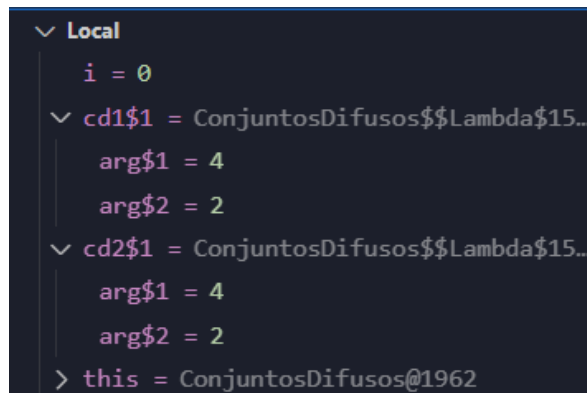


Figura 51: Debug inclusión

Condición else if ($pertenece(i, cd1) > pertenece(i, cd2)$): La función $pertenece(i, cd1)$ y $pertenece(i, cd2)$ se llama para obtener el valor de pertenencia del índice 0 en ambos conjuntos. Dado que $cd1$ y $cd2$ son el mismo conjunto (conjunto), el valor de pertenencia será idéntico. Por lo tanto, $pertenece(i, cd1) > pertenece(i, cd2)$ es falso, ya que el valor de pertenencia es igual para ambos conjuntos. Como la condición es falsa, la función llama recursiva mente a $loop(i + 1)$, incrementando i en 1.

```

Local
  i = 1
  cd1$1 = ConjuntosDifusos$$Lambda$153/0x0000...
    arg$1 = 4
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$153/0x0000...
    arg$1 = 4
    arg$2 = 2
  this = ConjuntosDifusos@1962

```

Figura 52: Debug inclusión

```

Local
  i = 2
  cd1$1 = ConjuntosDifusos$$Lambda$15...
    arg$1 = 4
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$15...
    arg$1 = 4
    arg$2 = 2
  this = ConjuntosDifusos@1962
    Class has no fields

```

Figura 53: Debug inclusión

```

Local
  i = 3
  cd1$1 = ConjuntosDifusos$$Lambda$15...
    arg$1 = 4
    arg$2 = 2
  cd2$1 = ConjuntosDifusos$$Lambda$15...
    arg$1 = 4
    arg$2 = 2
  this = ConjuntosDifusos@1962
    Class has no fields

```

Figura 54: Debug inclusión

```
▼ Local
  i = 4
  ▼ cd1$1 = ConjuntosDifusos$$Lambda$15...
    arg$1 = 4
    arg$2 = 2
  ▼ cd2$1 = ConjuntosDifusos$$Lambda$15...
    arg$1 = 4
    arg$2 = 2
  ▼ this = ConjuntosDifusos@1962
    Class has no fields
```

Figura 55: Debug inclusión

La función loop continúa incrementando i y repitiendo los mismos pasos hasta que $i = 1000$:

- Verifica if ($i > 1000$) (falso hasta llegar a 1001).
- Compara $\text{pertenece}(i, \text{cd1})$ y $\text{pertenece}(i, \text{cd2})$ en cada índice. Como cd1 y cd2 son el mismo conjunto, esta comparación siempre será falsa, y $\text{loop}(i + 1)$ se llama cada vez.

```
▼ Local
  i = 1000
  ▼ cd1$1 = ConjuntosDifusos$$Lambda$153/...
    arg$1 = 4
    arg$2 = 2
  ▼ cd2$1 = ConjuntosDifusos$$Lambda$153/...
    arg$1 = 4
    arg$2 = 2
  > this = ConjuntosDifusos@1962
```

Figura 56: Debug inclusión

Cuando i llega a 1001, se cumple la condición if ($i > 1000$), y loop retorna true. Esto indica que todos los elementos hasta el índice 1000 cumplen con la condición de inclusión (ningún valor de pertenencia en cd1 es mayor que el correspondiente en cd2).

```

▼ Local
  →loop$1() = true
  ▼ cd1 = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2
  ▼ cd2 = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2
  > this = ConjuntosDifusos@1962

```

Figura 57: Debug inclusión

```

▼ Local
  →loop$1() = true
  ▼ cd1 = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2
  ▼ cd2 = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2
  > this = ConjuntosDifusos@1962

```

Figura 58: Debug inclusión

La llamada a inclusion retorna true, y el assert en el test pasa satisfactoriamente. Este resultado confirma que conjunto está incluido en sí mismo, ya que la función inclusion ha verificado cada índice hasta 1000 y no ha encontrado discrepancias.

```

▼ Local
  →loop$1() = true
  ▼ cd1 = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2
  ▼ cd2 = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2
  > this = ConjuntosDifusos@1962

```

Figura 59: Debug inclusión

```

v Local
  -> inclusion() = true
  v $this = InclusionTest@2116 "InclusionTest"
    NoArgTest$module = null
  > objConjuntosDifusos = ConjuntosDifusos@1962
  > org$scalatest$funSuite$AnyFunSuiteLike$$eng:
  > styleName = "org.scalatest.FunSuite"
  > succeed = Succeeded$@2121 "Succeeded"
  v conjunto = ConjuntosDifusos$$Lambda$153/0x...
    arg$1 = 4
    arg$2 = 2

```

Figura 60: Debug inclusión

1.1.6. Función Igualdad

Esta función permite verificar si dos conjuntos difusos `cd1` y `cd2` son iguales. Para que dos conjuntos difusos sean considerados iguales, deben cumplir con dos condiciones:

- `cd1` debe estar incluido en `cd2`
- `cd2` debe estar incluido en `cd1` (es decir, debe cumplirse la inclusión en ambas direcciones).

En términos prácticos, esto implica que para cada valor en el rango, los valores de membresía en ambos conjuntos deben ser iguales. Si ambas inclusiones se cumplen, la función retorna `true`, de lo contrario retorna `false`.

Ejemplo para `igualdad(cd1, cd2)`

1. **Inicia la ejecución del test:** `test(igualdad - conjuntos con valores similares)`
2. **Primera llamada a `inclusion(cd1, cd2)`:**
 - Se llama a `inclusion(cd1, cd2)`.
 - Se invoca `loop(0)`.

```

v VARIABLES
  v Local
    s = 0
  > cd1$1 = IgualdadTest$$Lambda$146/0x000000...
  > cd2$1 = IgualdadTest$$Lambda$147/0x000000...
  > this = ConjuntosDifusos@1953

```

Figura 61: Debug igualdad

```

ConjuntosDifusos.loop$1(int,Function1,Function1): boolean  ConjuntosDifusos.sc...
ConjuntosDifusos.inclusion(Function1,Function1): boolean  ConjuntosDifusos.scala
ConjuntosDifusos.igualdad(Function1,Function1): boolean  ConjuntosDifusos.scala
IgualdadTest.$anonfun$new$1(IgualdadTest): Assertion     IgualdadTest.scala 16:1

```

Figura 62: Llamadas en pila

3. **assert(!objConjuntosDifusos.igualdad(cd1, cd2))**

La función **assert** invoca la función **igualdad(cd1, cd2)** para comprobar si los conjuntos son diferentes.

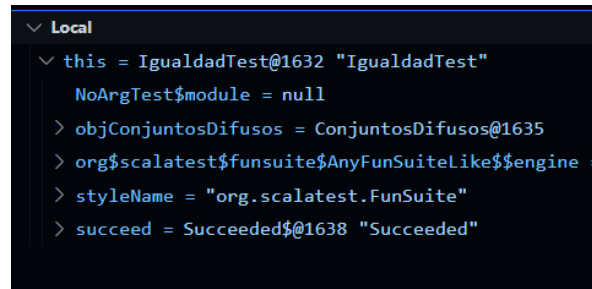


Figura 63: Debug Igualdad

4. **igualdad(cd1, cd2)**

Se ejecuta la función **igualdad** con los conjuntos **cd1** y **cd2** como parámetros.

5. **cd1**

La función **cd1** se invoca con el argumento 3, lo que retorna el valor 6, ya que **cd1** corresponde al conjunto difuso **grande(3, 6)**.

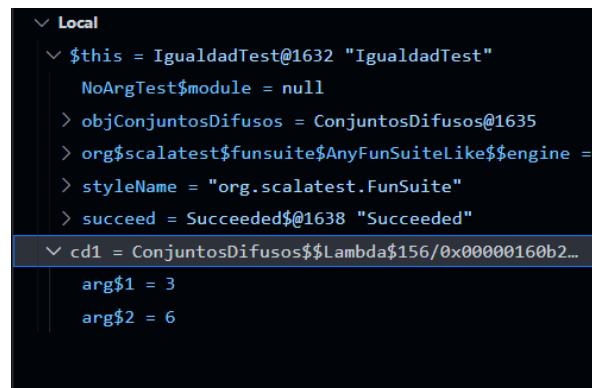


Figura 64: [H]

6. **cd2**

La función **cd2** se invoca con el argumento 3, lo que retorna el valor 7, ya que **cd2** corresponde al conjunto difuso **grande(3, 7)**.

```

  Local
  $this = IgualdadTest@1632 "IgualdadTest"
    NoArgTest$module = null
    > objConjuntosDifusos = ConjuntosDifusos@1635
    > org$scalatest$funsuite$AnyFunSuiteLike$$engine =
    > styleName = "org.scalatest.FunSuite"
    > succeed = Succeeded$@1638 "Succeeded"
  cd1 = ConjuntosDifusos$$Lambda$156/0x00000160b21...
    arg$1 = 3
    arg$2 = 6
  cd2 = ConjuntosDifusos$$Lambda$156/0x00000160b21...
    arg$1 = 3
    arg$2 = 7

```

Figura 65: Debug Igualdad

7. **Comparación: 6 == 7**

Se compara el valor obtenido de `cd1` (6) con el valor obtenido de `cd2` (7). La comparación es falsa.

8. **`igualdad(cd1, cd2)` retorna false**

Como la comparación entre 6 y 7 es falsa, la función `igualdad` retorna `false`.

9. **`assert(true)`**

La función `assert` verifica si el resultado de `igualdad(cd1, cd2)` es `true`. Dado que la comparación resultó en `false`, se aplica la negación y el test pasa correctamente, ya que la condición `assert(true)` se cumple.

```

  Local
  $this = IgualdadTest@1632 "IgualdadTest"
    NoArgTest$module = null
    > objConjuntosDifusos = ConjuntosDifusos@1635
    > org$scalatest$funsuite$AnyFunSuiteLike$$engine =
    > styleName = "org.scalatest.FunSuite"
    > succeed = Succeeded$@1638 "Succeeded"
  cd1 = ConjuntosDifusos$$Lambda$156/0x00000160b21...
    arg$1 = 3
    arg$2 = 6
  cd2 = ConjuntosDifusos$$Lambda$156/0x00000160b21...
    arg$1 = 3
    arg$2 = 7

```

Figura 66: Debug Igualdad

2. Informe de Corrección

2.1. Argumentación de Corrección

2.1.1. Función grande(d,e)

Se implementó la función grande() que recibe dos enteros, 'd' y 'e', y devuelve un conjunto difuso representado por la función 'evaluarGrande'. Sea este el siguiente programa en Scala:

```
1 def grande (d:Int, e:Int): ConjDifuso = {  
2   // Creamos una función interna que evaluar cada n mero  
3   def evaluarGrande(n: Int): Double = {  
4     // Para evitar divisi n por cero cuando n es negativo  
5     if (n < 0) 0.0  
6     else {  
7       // Calculamos (n/(n+d))^e  
8       val base = n.toDouble / (n + d).toDouble  
9       // Usamos Math.pow para elevar a la e  
10      Math.pow(base, e)  
11    }  
12  }
```

Esta función implementa el siguiente proceso iterativo para calcular el grado de pertenencia de un valor entero n en el conjunto difuso, utilizando los parámetros d y e .

- Un estado $s = (n, g)$ donde g representa el grado de pertenencia calculado.
- El estado inicial es $s_0 = (n, 0)$ si $n < 0$, y $s_0 = \left(n, \left(\frac{n}{n+d}\right)^e\right)$ si $n \geq 0$.
- s es el estado final, ya que la función evalúa el grado de pertenencia sin recurrir a nuevos llamados.
- La invariante de ciclo es $\text{Inv}(s) \equiv n < 0 \Rightarrow g = 0,0 \wedge n \geq 0 \Rightarrow g = \left(\frac{n}{n+d}\right)^e$.

Demostración

1. **Inv(s_0):** el estado inicial cumple la condición invariante.

Para $n < 0$, se cumple que $g = 0,0$. Para $n \geq 0$, se cumple que $g = \left(\frac{n}{n+d}\right)^e$.

2. **La invariante cumple la transformación de estados.**

Dado que no hay un ciclo en esta función, no hay una transformación de estados repetitiva. Sin embargo, podemos verificar que en cualquier llamado de *evaluarGrande*, si $n \geq 0$, se cumple directamente la invariante mediante el cálculo de $g = \left(\frac{n}{n+d}\right)^e$. Si $n < 0$, la función retorna 0,0, manteniendo la invariante.

3. **Inv(s_f) implica la respuesta correcta.**

Dado que el estado final s_f en cada llamado de *evaluarGrande* devuelve un valor de g tal que $g = 0,0$ si $n < 0$ y $g = \left(\frac{n}{n+d}\right)^e$ si $n \geq 0$, la función cumple con el cálculo esperado para el conjunto difuso.

Esto implica que la función grande(d, e) cumple correctamente con la especificación del conjunto difuso, retornando una función que calcula el grado de pertenencia conforme a los valores de d y e .

2.1.2. Función complemento(c)

La función complemento recibe un conjunto difuso c y devuelve su complemento, calculando el valor de pertenencia inverso para cada elemento. Para lograr esto, se define una función interna llamada calcularComplemento que calcula el complemento de cada valor de pertenencia como $1 - c(x)$, donde $c(x)$ es el valor de pertenencia original en c . Finalmente, esta función es retornada como la representación del conjunto complementario, asegurando que los elementos con valores de pertenencia altos en c tengan valores bajos en el complemento, y viceversa.

```

1 def complemento(c: ConjDifuso): ConjDifuso = {
2   // Creamos una funci n que calcula el complemento
3   def calcularComplemento(x: Int): Double = {
4     // El complemento es 1 - f(x)
5     1.0 - c(x)
6   }
7
8   // Retornamos la funci n que representa el complemento
9   calcularComplemento
10 }

```

Esta función implementa el siguiente proceso iterativo para calcular el complemento de un conjunto difuso c , devolviendo el valor de pertenencia inverso $1 - c(x)$ para cada elemento x en el conjunto.

- Un estado $s = (n, g)$ donde g representa el grado de pertenencia calculado en el complemento.
- El estado inicial es $s_0 = (n, 1 - c(n))$, donde n representa el valor entero en el conjunto difuso original c y $c(n)$ es su valor de pertenencia en c .
- s es el estado final, ya que la función evalúa el grado de pertenencia en el complemento sin recurrir a nuevos llamados.

Demostración

1. **Invariancia de estado:** Para cada valor n , la función `complemento` calcula correctamente $1 - c(n)$ como el valor de pertenencia en el complemento, cumpliendo con la definición del complemento en conjuntos difusos.
2. **Corrección de la función:** Dado que la función `calcularComplemento` devuelve correctamente $1 - c(x)$ para cada elemento x , la función cumple con la especificación para conjuntos difusos complementarios.

2.1.3. Función union(c)

La función `union` toma dos conjuntos difusos, `cd1` y `cd2`, y devuelve un nuevo conjunto difuso que representa la unión de ambos. La función utiliza una función interna `calcularUnion` que, para cada elemento n , calcula el valor de pertenencia en la unión de los conjuntos difusos como el valor máximo entre las pertenencias de `cd1` y `cd2` en dicho elemento. A continuación, mostramos el código:

```

1 // Creamos una funci n que calcula la uni n de dos conjuntos difusos
2 def calcularUnion(n: Int): Double = {
3   // La uni n es el m ximo de las pertenencias
4   math.max(cd1(n), cd2(n))
5 }
6
7 // Retornamos la funci n que representa la uni n
8 calcularUnion
9 }

```

Este programa implementa el siguiente proceso iterativo:

- Un estado es $s = (n, cd1, cd2)$
- El estado inicial es $s_0 = (n, cd1, cd2)$
- $(n, cd1, cd2)$ es final si el valor de pertenencia ha sido calculado, es decir, si `calcularUnion(n)` ha evaluado `math.max(cd1(n), cd2(n))`.
- La invariante de ciclo es `calcularUnion(n) = máx(cd1(n), cd2(n))`. La invariante de ciclo es una relación que **siempre** se cumple en el cálculo.
- `transformar(n, cd1, cd2) = (n, cd1, cd2)`

Demostración

1. **Inv**(s_0): el estado inicial cumple la condición invariante.

$$s_0 = (n, cd1, cd2) \Rightarrow \text{calcularUnion}(n) = \text{máx}(cd1(n), cd2(n))$$

2. La invariante cumple la transformación de estados ($s_i \neq s_f \Rightarrow \text{Inv}(\text{transformar}(s_i))$)
 - Dado que no hay un cambio de estado en cada evaluación de **calcularUnion**, la invariante se mantiene.
3. **Inv**(s_f) implica respuesta(s_f) = f(cd1, cd2)

$$\text{calcularUnion}(n) = \text{máx}(cd1(n), cd2(n))$$

Con esto, hemos demostrado que la función **union** es correcta, ya que la invariante de ciclo garantiza que el valor devuelto por **calcularUnion** representa la unión de los conjuntos difusos según la definición esperada.

2.1.4. Función interseccion(c)

La función **interseccion** toma dos conjuntos difusos, **cd1** y **cd2**, y devuelve un nuevo conjunto difuso que representa la intersección de ambos. La función utiliza una función interna **calcularInterseccion** que, para cada elemento n , calcula el valor de pertenencia en la intersección de los conjuntos difusos como el valor mínimo entre las pertenencias de **cd1** y **cd2** en dicho elemento. A continuación, mostramos el código:

```

1 // Creamos una función que calcula la intersección de dos conjuntos difusos
2 def calcularInterseccion(n: Int): Double = {
3     // La intersección es el mínimo de las pertenencias
4     math.min(cd1(n), cd2(n))
5 }
6
7 // Retornamos la función que representa la intersección
8 calcularInterseccion
9 }
```

Este programa implementa el siguiente proceso iterativo:

- Un estado es $s = (n, cd1, cd2)$
- El estado inicial es $s_0 = (n, cd1, cd2)$
- $(n, cd1, cd2)$ es final si el valor de pertenencia ha sido calculado, es decir, si **calcularInterseccion**(n) ha evaluado **math.min(cd1(n), cd2(n))**.
- La invariante de ciclo es **calcularInterseccion**(n) = mín(cd1(n), cd2(n)). La invariante de ciclo es una relación que **siempre** se cumple en el cálculo.
- **transformar**($n, cd1, cd2$) = ($n, cd1, cd2$)

Ahora, demostramos los puntos mencionados:

1. **Inv**(s_0): el estado inicial cumple la condición invariante.

$$s_0 = (n, cd1, cd2) \Rightarrow \text{calcularInterseccion}(n) = \text{mín}(cd1(n), cd2(n))$$

2. La invariante cumple la transformación de estados ($s_i \neq s_f \Rightarrow \text{Inv}(\text{transformar}(s_i))$)
 - Dado que no hay un cambio de estado en cada evaluación de **calcularInterseccion**, la invariante se mantiene.
3. **Inv**(s_f) implica respuesta(s_f) = f(cd1, cd2)

$$\text{calcularInterseccion}(n) = \text{mín}(cd1(n), cd2(n))$$

Con esto, hemos demostrado que la función **interseccion** es correcta, ya que la invariante de ciclo garantiza que el valor devuelto por **calcularInterseccion** representa la intersección de los conjuntos difusos según la definición esperada.

2.1.5. Función `inclusion(cd1, cd2)`

Se implemento la Función **`inclusion()`** que recibe dos conjuntos difusos `cd1` y `cd2` y verifica si todos los elementos de `cd1` están incluidos en `cd2`. Formalmente, esto significa verificar que:

$$cd_1 \subseteq cd_2 \quad \text{si y solo si} \quad \forall s \in \mathbb{Z}, cd_1(s) \leq cd_2(s)$$

Es decir, para cada valor s , la membresía de $cd_1(s)$ debe ser menor o igual que la membresía de $cd_2(s)$. Si para algún s se cumple que $cd_1(s) > cd_2(s)$, entonces cd_1 **no está incluido** en cd_2 .

Análisis de la implementación

La implementación de esta función en Scala se realiza mediante una función auxiliar recursiva `loop(s)`, que evalúa la inclusión para cada valor s en el rango de 0 a 1000. La función actúa de la siguiente forma:

1. **Condición de parada:** Si $s > 1000$, la función retorna **`true`**, indicando que $cd_1 \subseteq cd_2$ en todo el dominio evaluado.
2. **Condición de inclusión:** Para cada valor s , la función compara $cd_1(s)$ y $cd_2(s)$. Si encuentra que $cd_1(s) > cd_2(s)$ para algún s , la función retorna **`false`**, indicando que cd_1 no está incluido en cd_2 .
3. **Llamada recursiva:** Si no se cumple ninguna de las condiciones anteriores, la función llama recursivamente a `loop(s + 1)`, evaluando el siguiente valor.

Sea esta el siguiente programa en Scala:

```
1 def inclusion(cd1: ConjDifuso, cd2: ConjDifuso): Boolean = {
2   @scala.annotation.tailrec
3   def loop(s: Int): Boolean = {
4     if (s > 1000) true // Finaliza cuando se recorren todos los elementos hasta
5     1000
6     else if (pertenece(s, cd1) > pertenece(s, cd2)) false // Si no cumple la
7     inclusion retorna false
8     else loop(s + 1)
9   }
10  loop(0)
11 }
```

Para demostrar que la función es correcta, utilizaremos inducción sobre el índice s :

- **Caso base:** $s = 1001$

$$P_f(1001) \rightarrow \text{true}$$

Cuando $s > 1000$, la función devuelve **`true`**, lo que indica que todos los elementos de $cd1$ hasta el índice 1000 están incluidos en $cd2$. Esto es consistente con la definición de inclusión en conjuntos difusos en todo el dominio evaluado.

- **Caso de inducción:** Supongamos que la hipótesis de inducción (HI) es cierta para $s = k$, con $0 \leq k \leq 1000$.

$$P_f(k) = \text{true} \Rightarrow \text{pertenece}(k, cd1) \leq \text{pertenece}(k, cd2)$$

Ahora consideramos el caso $s = k + 1$:

$$P_f(k + 1) \rightarrow \begin{cases} \text{false}, & \text{si } \text{pertenece}(k + 1, cd1) > \text{pertenece}(k + 1, cd2) \\ \text{loop}(k + 2), & \text{si } \text{pertenece}(k + 1, cd1) \leq \text{pertenece}(k + 1, cd2) \end{cases}$$

Usando la hipótesis de inducción (HI), sabemos que si:

$$\text{pertenece}(k + 1, cd1) \leq \text{pertenece}(k + 1, cd2)$$

$$\text{entonces} \Rightarrow P_f(k + 1) = f(cd1, cd2)$$

Por lo tanto, $P_f(k + 1)$ es correcto siempre que se cumpla la relación $cd_1(s) \leq cd_2(s)$ para cada s en el dominio evaluado.

Conclusión

Concluimos que, por inducción, $\forall \text{ ConjDifuso } cd1, cd2, P_f(cd1, cd2) = f(cd1, cd2)$. Por lo tanto la implementación es correcta, ya que cumple con la definición matemática de inclusión de conjuntos difusos. La función evalúa para cada s si $cd1(s) \leq cd2(s)$ y, si encuentra una violación, termina la recursión retornando **false**. En caso contrario, retorna **true** al recorrer el dominio completo. La estructura recursiva y las condiciones de parada garantizan una verificación exhaustiva de la inclusión en el dominio especificado.

2.1.6. Función igualdad(cd1,cd2)

```
1 // Funci n para verificar si dos conjuntos difusos son iguales
2 def igualdad(cd1: ConjDifuso, cd2: ConjDifuso): Boolean = {
3     inclusion(cd1, cd2) && inclusion(cd2, cd1)
4 }
5 }
```

La función **igualdad** tiene como objetivo verificar si dos conjuntos difusos cd_1 y cd_2 son **iguales**. La definición matemática de igualdad para conjuntos difusos establece que dos conjuntos cd_1 y cd_2 son iguales si, y solo si, cumplen las siguientes dos condiciones de **inclusión mutua**:

$$cd_1 = cd_2 \quad \text{si y solo si} \quad (cd_1 \subseteq cd_2) \text{ y } (cd_2 \subseteq cd_1)$$

Esto significa que para que cd_1 y cd_2 sean iguales, ambos deben estar mutuamente incluidos, es decir:

1. $cd_1 \subseteq cd_2$
2. $cd_2 \subseteq cd_1$

Análisis de la implementación

La implementación de la función **igualdad** en Scala se basa en la comprobación de estas inclusiones mutuas:

```
def igualdad(cd1: ConjDifuso, cd2: ConjDifuso): Boolean = {
    inclusion(cd1, cd2) && inclusion(cd2, cd1)
}
```

Esta implementación utiliza la función **inclusion** para verificar si ambos conjuntos cumplen con las condiciones de inclusión mutua. Si ambas inclusiones son verdaderas, entonces los conjuntos son iguales y la función devuelve **true**; de lo contrario, devuelve **false**.

Notación matemática de la implementación

La implementación se puede expresar matemáticamente como sigue:

1. Inclusión de cd_1 en cd_2 :

La primera condición, $cd_1 \subseteq cd_2$, se verifica mediante la función **inclusion(cd1, cd2)**. Matemáticamente, esto significa que:

$$\forall s \in \mathbb{Z}, cd_1(s) \leq cd_2(s)$$

2. Inclusión de cd_2 en cd_1 :

La segunda condición, $cd_2 \subseteq cd_1$, se verifica mediante la función **inclusion(cd2, cd1)**. Matemáticamente, esto significa que:

$$\forall s \in \mathbb{Z}, cd_2(s) \leq cd_1(s)$$

3. Condición de igualdad:

La función `igualdad` retorna `true` si **ambas** inclusiones se cumplen, es decir:

$$(\forall s \in \mathbb{Z}, cd_1(s) \leq cd_2(s)) \quad \text{y} \quad (\forall s \in \mathbb{Z}, cd_2(s) \leq cd_1(s))$$

Si ambas condiciones se cumplen, entonces $cd_1 = cd_2$. De lo contrario, retorna `false`.

Verificación de la corrección

■ Caso 1: Los conjuntos son iguales.

Si $cd_1(s) = cd_2(s)$ para todos los $s \in \mathbb{Z}$, entonces ambas inclusiones se cumplen automáticamente, ya que:

$$cd_1(s) \leq cd_2(s) \quad \text{y} \quad cd_2(s) \leq cd_1(s) \quad \forall s$$

En este caso, la función `igualdad` retornará `true`, lo cual es correcto.

■ Caso 2: Los conjuntos no son iguales.

Si existe algún $s_0 \in \mathbb{Z}$ tal que $cd_1(s_0) > cd_2(s_0)$, entonces cd_1 no está incluido en cd_2 , y la función `inclusion(cd1, cd2)` retornará `false`. Esto implica que la función `igualdad` retornará `false`, lo cual es correcto.

Similarmente, si existe algún $s_1 \in \mathbb{Z}$ tal que $cd_2(s_1) > cd_1(s_1)$, entonces cd_2 no está incluido en cd_1 , y la función `inclusion(cd2, cd1)` retornará `false`. En este caso, la función `igualdad` también retornará `false`, lo cual es correcto.

■ Caso 3: Diferencia en algunos valores pero no en todos.

Si existe alguna diferencia en los valores de membresía entre cd_1 y cd_2 , pero no se cumple la inclusión mutua, la función `igualdad` retornará `false`, lo cual es correcto.

Conclusión sobre la corrección

La implementación de la función `igualdad` es correcta, ya que cumple con la definición matemática de igualdad para conjuntos difusos. La función verifica que ambos conjuntos cd_1 y cd_2 sean mutuamente inclusivos, lo cual es la condición necesaria y suficiente para que sean iguales. La estructura del código y el uso de la función `inclusion` garantizan que el algoritmo se comporta correctamente en todos los casos posibles de igualdad de conjuntos difusos.

3. Conclusiones

Al realizar este taller de funciones de alto orden, hemos implementado y comprobado el comportamiento de conjuntos difusos basados en funciones características, según la teoría de Lofti Zadeh. Utilizando técnicas de programación funcional en Scala, hemos logrado modelar estos conjuntos y realizar operaciones como conjunto difuso de números grandes, complemento, inclusión, igualdad, unión e intersección.

- **Importancia del grado de pertenencia:** El grado de pertenencia en los conjuntos difusos es crucial porque facilita la categorización de los elementos, ofreciendo mayor flexibilidad y precisión en la representación de la realidad, esto permite combinar y manipular funciones de pertenencia de manera más efectiva.
- **Pruebas y Verificación:** Los casos de prueba y ejemplos han confirmado la correcta implementación de las operaciones de conjuntos difusos, siguiendo los principios de la teoría de conjuntos difusos de Lofti Zadeh. Esto permite modelar problemas reales donde la pertenencia no siempre es clara, como conjuntos de números grandes o con límites difusos.

- **Eficiencia de los Algoritmos:** Analizar la eficiencia de los algoritmos en términos de tiempo de ejecución y uso de recursos es fundamental. Evaluar cuándo utilizar recursión o iteración puede optimizar el rendimiento. La recursión es útil para resolver problemas complejos, mientras que la iteración puede ser más eficiente en casos donde se necesita un control explícito del flujo.
- **Abstracción y Modularidad:** funciones de alto orden permiten una mayor abstracción y modularidad en el código. Esto facilita la separación de preocupaciones y la reutilización de componentes, lo que es particularmente útil al trabajar con conjuntos difusos donde las operaciones pueden ser complejas y variadas.